Lesson 3
Understanding General Software Development

Learning Objectives

Students will learn about:

- Application Lifecycle Management
- Testing
- Data Structures
- Sorting Algorithms

You are a software developer for the Northwind Corporation. You work as part of a team to develop computer programs that solve complex business problems. As a developer, you need to be aware of the different phases of the application lifecycle because you play an important role in multiple parts of this cycle. For instance, not only do you participate in the design and development portions of the cycle, but you often need to interact with the software testing team during the testing portion of the cycle. Sometimes, you even engage in testing yourself, so you need to have a general understanding of this process

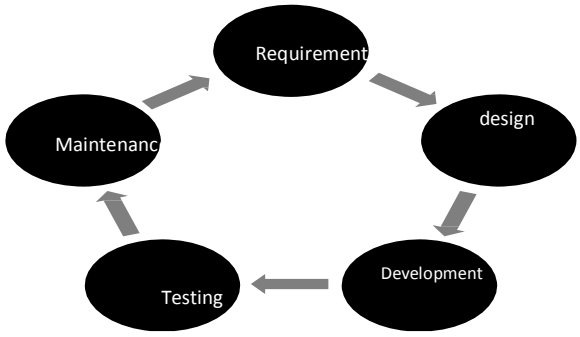
When you develop software, you use various types of data structures and algorithms. Therefore, you need to know which data structure to use for the task at hand and what the performance implications of your choice are. You should also have a general understanding of various sorting methods.

OBJECTIVE1: Understanding Application Lifecycle Management

Application lifecycle management (ALM) is the set of activities that revolve around a new software product, from its inception to when the product matures and perhaps retires

Developing a software application involves more than just writing the code. Various other activities also need to be performed in the right order to develop a successful application. Collectively, these activities are known as application lifecycle management (ALM). Some of the activities that are part of the ALM process are shown in Figure 3-1, including requirements, design, development, testing, delivery, and release management.

Figure3-1



In this section, you'll learn about the different activities and roles involved in each stage of the ALM process.

The application lifecycle starts when the need for a new software application is identified.

A business manager is usually the person who is the sponsor of the project. He or she analyzes the need, checks how the project fits with the overall strategy of the business, arranges the funding, and initiates the staffing process for the project.

A project manager is probably the first person hired by the business manager. The project manager is responsible for the overall execution of the project. His or her key responsibilities are to make sure that the project stays on budget and finishes on time. The project manager is also responsible for hiring team members and for facilitating cooperation within the team.

Understanding Requirements Analysis

Requirements analysis is one of the most important steps in the application lifecycle. Precise, complete, well-documented requirements are critical to the success of the project. These requirements can be functional or nonfunctional. Functional requirements specify exactly what the system is designed to accomplish. In contrast, nonfunctional requirements are quality requirements such as scalability, security, reliability, and so on

A business analyst is responsible for analyzing business needs and converting them into requirements that can be executed by the development team.

Understanding the Design Process

The design process is used to create plans, models, and architecture for how the software will be implemented.

The design process generates detailed technical specifications that will be used for developing the

system. The output of the design process is a set of technical models and specifications that provide guidance to the developers and other team members during the software development activity. The output of the design process is more abstract than concrete. At this point, no real system exists that you can interact with.

Some of the most important participants in this stage of the ALM process include an architect and a user-experience designer:

- Architect: An architect designs the technical blueprint of the system. This includes identifying components and services, their behavior, and how they interact with each other and with the external world.
- User-experience designer: A user-experience designer creates the user experience of the system. This includes designing the user interface (UI) elements; designing navigation between various forms, screens, or pages; and so on.

Understanding Software Development

The software development activity involves implementing design by creating software code, databases, and other related content.

Software development is the portion of the ALM process in which the business requirements are implemented in working code based on the design that was created in the previous activity. At the end of this activity, you have concrete output in form of a software system with which users can interact.

Critical participants in software development include the following:

- Developers: Developers write code based on the requirements gathered by the business analyst, the architecture laid down by the architect, and the user experience developed by the user-experience designer.
- Database administrators (DBAs): DBAs are responsible for implementation and maintenance of the software's databases. DBAs also plan for data integrity, security, and speed.
- Technical writers: Technical writers develop the system manuals and help files that will be delivered along with the application.
- Content developers: Content developers are subject matter experts who develop the content for the system. For example, if the application is a movie review website, just deploying the website is not enough—you also need to make sure that the site has enough content to gather user interest.

Understanding Software Testing

Software testing verifies that the implementation matches the requirements of the system.

Software testing is used to assure the quality of the final product. Testing can identify possible gaps between the system expectations described in the requirements document and actual system behavior.

Among the most critical participants in the software testing activity are the testers who verify the working application to make sure that it satisfies the identified requirements. When these testers identify any defects in the application, they assign each defect to an appropriate person who can fix it. For example, a code defect would be assigned back to a developer so he or she could remedy the error.

Understanding Release Management

The release management activity is used to manage the deployment, delivery, and support of software releases

Release management includes activities such as packaging and deploying the software, managing software defects, and managing software change requests.

Major players in the release management activity include the following individuals:

- Release manager: The release manager coordinates various teams and business units to ensure timely release of a software product.
- Operation staff: The operation staff members make sure that the system is delivered as promised. This could involve burning DVDs and shipping them as orders are received, or it could entail maintaining a Software as a Service (SaaS) system on an ongoing basis. Operation staff are also responsible for releasing any system updates (e.g., bug fixes or new features).
- Technical support staff: These staffers interact with customers and help solve their problems with the system. Technical support can generate valuable metrics about what areas of the system are most difficult for users and possibly need to be updated in the next version of the application.

OBJECTIVE2: Understanding Testing

Software testing is the process of verifying software against its requirements. Testing takes place after most development work is completed

As previously mentioned, software testing is the process of verifying that a software application works as expected and fulfills all its business and technical requirements. When there is a difference between the expected behavior and the actual behavior of the system, a software defect (or “bug”) is logged and eventually passed on to an individual who is responsible for fixing it.

Software testing may involve both functional and nonfunctional testing. Functional testing relates to the functional requirements of the system, and it tests those features that make up the core functionality of the system. For example, testing whether users can add items to a shopping cart is an important part of functional testing for an e-commerce Web site. In comparison, nonfunctional testing involves testing software attributes that are not part of the core functionality but rather part of the software's nonfunctional requirements, such as scalability, usability, security.

TakeNote

It is important to note that the process of software testing can only help find defects—it cannot guarantee the absence of defects. Complex software has a huge number of possible execution paths and many parameters that can affect its behavior. It is not feasible and often not possible to test all the different situations that such software will encounter in a production environment.

Understanding Testing Methods

Software testing methods are generally divided into two categories: white-box and black- box testing Traditionally, there are two broad approaches to software testing:

- Black-box testing
- White-box testing

Black-box testing treats the software as a black box, focusing solely on inputs and outputs. With this approach, any knowledge of internal system workings is not used during testing. In contrast, with white-box testing, testers use their knowledge of system internals when testing the system. For example, in white-box testing, the testers have access to the source code.

These two testing techniques complement each other. Black-box testing is mostly used to make sure a software application covers all its requirements. Meanwhile, white-box testing is used to make sure that each method or function has proper test cases available.
Understanding Testing Levels

Testing is performed at various phases of the application development lifecycle. Different testing levels specify where in the lifecycle a particular test takes place, as well as what kind of test is being performed

Testing levels are defined by where the testing takes place within the course of the software development lifecycle. Five distinct levels of testing exist:

- **Unit testing:** Unit testing verifies the functionality of a unit of code. For example, a unit test may assess whether a method returns the correct value. Unit testing is white-box testing, and it is frequently done by the developer who is writing the code. Unit testing often uses an automated tool that can simplify the development of cases and also keep track of whether a

code modification causes any of the existing unit tests to fail. Visual Studio has built-in support for unit testing. You can also use open-source tools such as NUnit to automate unit tests for the .NET Framework code.

- **Integration testing**: Integration testing assesses the interface between software components. Integration testing can be performed incrementally as the components are being developed, or it can be performed as a "big bang" when all the components are ready to work together. The former approach is preferred to the latter because it reduces risk and increases stakeholders' confidence as the system is being developed. Integration testing can also involve testing the component's interaction with an external system. For example, if a component relies on data from an external Web service, integration testing ensures that the component is working well with the external application.

- **System testing:** System testing is the overall testing of the software system. At this point, all the system components are developed and are working together and with any external systems.

- **Acceptance testing**: This level of testing is often performed by the customers themselves. There are generally two levels of acceptance testing prior to broad release of a product: alpha testing and beta testing. Alpha testing is performed by a limited group of users, and it is an opportunity to provide an early look at the product to the most important customers and gather feedback. Alpha releases may miss some features and generally lack many nonfunctional attributes such as performance. In the next level of testing, beta testing, you release the product to a wider audience of customers and solicit feedback. In terms of functionality, the beta release of the software is very close to the final release. However, the development teams might still be working on improving performance and fixing known defects.

- **Regression testing:** As the defects in a software application are reported and fixed, it is important to make sure that each new fix doesn't break anything that was previously working. This is where regression testing comes in handy. With every new fix, software testers usually run a battery of regression tests to make sure that each functionality that was already known to work correctly is still working

TakeNote

It is much more cost- effective to find defects earlier (rather than later) in the product development cycle

Understanding Data Structures

Data structures are techniques for organizing and storing data in computer memory. How the data is stored affects how the data is retrieved and manipulated. Understanding a data structure involves not only understanding the storage pattern, but also knowing what methods are used to create, access, and manipulate the data structure.

Data structures are the building blocks of most computer programs, and they allow developers to implement complex functionality. Most programming frameworks provide built-in support for a variety of data structures and associated methods to manipulate these data structures. In this section, you will learn about several distinct types of data structures so that you are familiar with the general techniques for manipulating them

Understanding Arrays

An array is a collection of items stored in a contiguous memory location and addressed using one or more indices

An array is a common data structure that represents a collection of items of a similar type. The items in an array are stored in contiguous memory locations. An array is a homogeneous data structure because the all the items of an array are of the same data type. Any array item can be directly accessed by using an index. In .NET Framework, array indexes are zero-based.

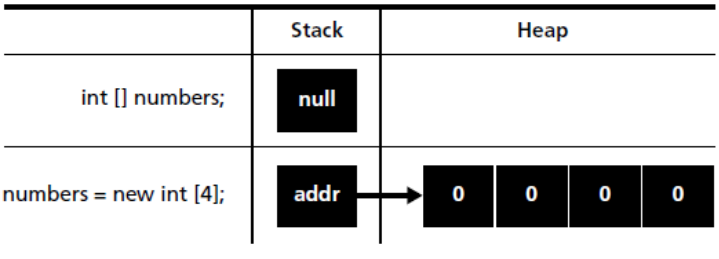
INTERNAL REPRESENTATION

In the following code, the first statement creates an array variable, and the second statement initializes the variable with an array of four integers:

```
int[] numbers;
 numbers = new int[4];
```

At first, the variable numbers are set to null because the array is not yet initialized. However, the second statement initializes the array by allocating a contiguous memory space big enough to store four integers in the memory heap. The starting address in the memory allocation is stored in the array variable numbers, as shown in Figure 3-2. All the array elements are initialized in this case with the value 0 because 0 is the default value for an integer

Figure 3-2
Internal representation of an array data structure



The variable numbers then acts as a reference to the memory location assigned to the array. The array name can be used to access each of the array items directly. In the .NET Framework, all arrays are zero-

based—that is, the first item of the array is accessed using an index of numbers[0], the second item is accessed by numbers[1], and so on.

It is also possible to have multidimensional arrays. A two-dimensional array can be thought of as a table in which each cell is an array element and can be addressed using the numbers of the row and column to which it belongs. Both the row number and column number are indexed by zero.

COMMON OPERATIONS
Arrays support the following operations:

- Allocation
- Access

To work with an array, you first allocate the memory by creating and initializing the array, as shown previously. Once the array is allocated, you can access any array element in any order you please by directly referring to its index. For example, the following code assigns a value of 10 to the fourth item of the array, and twice that value is then assigned to the variable calc:

```
number[3] = 10;
int calc = number[3] * 2;
```

PERFORMANCE AND USAGE
The contents of an array are laid out as a contiguous block of memory and can be accessed directly by using the array index. Thus, reading from or writing to an array is extremely fast. However, arrays are limited by the requirements of homogeneity and fixed size. Although array size can be increased, doing so requires reallocation of all the array elements and is a time-consuming operation.

Arrays work best when the number of items in the collection is predetermined and fast, direct access to each item is required.

In the .NET Framework, you can use the ArrayList class to get around an array's require- ments for homogeneity and fixed size. An ArrayList is a collection type that can hold items of any data type and dynamically expand when needed. However, an ArrayList is not as fast as an array.

Understanding Queues

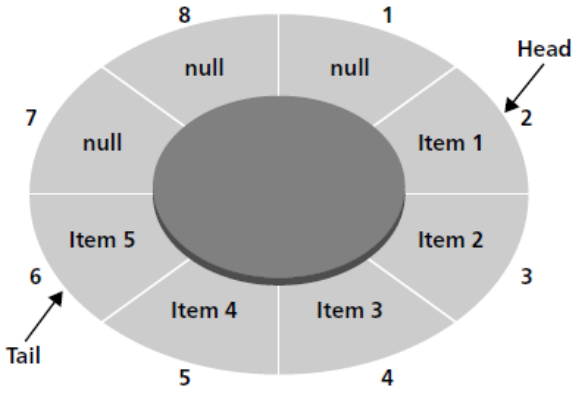
A queue is a collection of items in which the first item added to the collection is the first one to be removed.

The queue data structure mimics a real-life queue. In a queue, items are processed in the order in which they were added to the queue. In particular, items are always added at the end of the queue and removed from the front of the queue. This is also commonly known as first-in, first-out (FIFO) processing. The capacity of a queue is the number of items the queue can hold. However, as elements are added to

the queue, the capacity is automatically increased.
A queue is also a heterogeneous data structure, meaning that items in a queue can be of different data types.

INTERNAL REPRESENTATION
In order to avoid excessive reallocation of memory space and allow easy management, a queue is often internally implemented as a circular array of objects, as shown in Figure 3-3.

Figure 3-3
Internal representation of a queue data structure



Within a queue, the head index points to the first item, and the tail index points to the last item. In Figure 3-3, for example, the head index points to location 2 on the queue. Because the queue is circular, as long as you can keep track of the head and tail pointers, it doesn't matter what location the queue starts from. When an item is removed, the head moves to the next item in the queue. When a new item is added, it always appears at the end of the queue, and the tail starts pointing to the newly added item. Any null slots in a queue (including the one depicted in Figure 3-3) are the empty spots that can be filled before the queue will require a memory reallocation.

The.NET Framework provides an implementation of the queue data structure as part of the Queue class in the System.Collections namespace. In programming languages that don't provide an implementation of a queue, you can write your own Queue class by using an array-like data structure and simulating the queue operations

TakeNote

A generic version of the Queue class is available as part of the System. Collections.Generic namespace. This generic version is used to create a queue of items that are of the same data type.

COMMON OPERATIONS
A queue supports the following common operations:

- Enqueue: The enqueue operation first checks whether there is enough capacity available in the queue to add one more item. If capacity is available, the item is added to the tail end of the queue. If there is no space available in queue, the array is reallocated by a prespecified growth factor, and the new item is then added to the queue.

- Dequeue: The dequeue operation removes the current element at the head of the queue and sets the head to point to the next element.
- Peek: The peek operation allows you to look at the current item at the head position without actually removing it from the queue.
- Contains: The contains operation allows you to determine whether a particular item exists in the queue.

PERFORMANCE AND USAGE

A queue is a special-purpose data structure that is best suited for an application in which you need to process items in the order they were received. Some examples may include print spoolers, messaging systems, and job schedulers. Unlike an array, a queue cannot be used to randomly access elements. Operations such as enqueue and dequeue actually add and remove the items from the queue.

Understanding Stacks

A stack is a collection of items in which the last item added to the collection is the first one to be removed

As opposed to a queue, a stack is a last-in, first-out (LIFO) data structure. Think of a stack as similar to a stack of dinner plates on a buffet table; here, the last plate to be added is also the first plate to be removed. The capacity of a stack refers to the number of items it can hold. However, as elements are added to a stack, the stack's capacity is automatically increased.
A stack is a heterogeneous data structure, meaning that the items within it can be of different data types.

INTERNAL REPRESENTATION
Like a queue, a stack is often implemented as a circular buffer in order to avoid excessive reallocation of memory space and permit easier management. A stack can be visualized just like the queue shown in Figure 3-3, except that the tail is now called the top of the stack and the head is now called the bottom of the stack.

New items are always added to the top of a stack; when this happens, the top of the stack starts pointing to the newly added element. Items are also removed from the top of the stack, and when that happens, the top of the stack is adjusted to point to the next item in the stack.

The .NET Framework provides an implementation of the stack data structure as part of the Stack class in the System.Collections namespace. In programming languages that don't provide an implementation of the stack, you can write your own Stack class by using an array-like data structure and simulating the stack operations

TakeNote

A generic version of the Stack class is available as part of the System. Collections.Generic namespace. This generic version is used to create a stack of items that are of the same data type

COMMON OPERATIONS

A stack supports the following common operations:

- Push: The push operation first checks whether there is enough capacity available in the stack to add one more item. If capacity is available, the item is added to the top of the stack. If there is no space in the stack, the array is reallocated by a prespecified growth factor, and then the new item is added to the stack.
- Pop: The pop operation removes the element at the top of the stack and sets the top to point to the next element in the stack.
- Peek: The peek operation allows you to look at the current item at the top of the stack without actually removing it from the stack.
- Contains: The contains operation allows you to determine whether a particular item exists in the stack.

PERFORMANCE AND USAGE

A stack is a special-purpose data structure that is best suited for applications in which you need to process items in last-in, first-out order. Stack are useful structures because of their applications in runtime memory management, expression evaluation, method-call tracking, etc. Unlike an array, a stack cannot be used to access elements randomly. Operations such as push and pop actually add and remove the items from the stack.

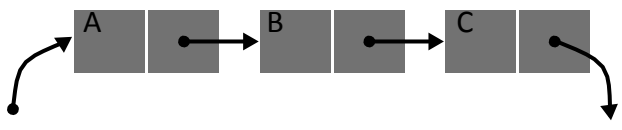
Understanding Linked Lists

A linked list is a collection of nodes arranged so that each node contains a link to the next node in the sequence.

A linked list is a collection of nodes in which each node contains a reference (or link) to the next node in the sequence. Unlike an array, the items in a linked list need not be contiguous; therefore, a linked list does not require reallocation of memory space for the entire list when more items must be added.

INTERNAL REPRESENTATION

In memory, a linked list can be visualized as a collection of nodes, as shown in Figure 3-4.



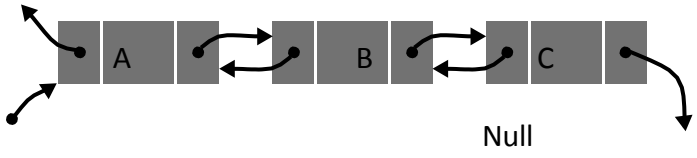
Head Null

Each node in a linked list contains of two pieces of information: the data corresponding to the node, and the link to the next node. The first node of the list is called the head node. Using the link in the head node, you can get to the next node and continue traversing nodes until the final link is a null value. Often, the term tail is used to refer to the list pointed to by the head node—that is, it refers to everything after the head node. Thus, in Figure 3-4, the tail is the linked list starting from node B.

Several other implementations of linked lists may also be used depending on requirements. For instance, in a circular linked list, the last node in the list points back to the first node to create a circle. In contrast, in a doubly linked list, each node contains two links, as shown in Figure 3-5.

Figure 3-5

Null



Head

At each node of a doubly linked list, one link is a forward reference that points to the next node in the sequence, and the other link is a backward reference that points to the previous node in the sequence. As you can imagine, a doubly linked list is easy to traverse in either direction.

The .NET Framework provides a LinkedList class as part of the System.Collections.Generic namespace. This class implements a homogeneous doubly linked list of the specified data type. You can also write your own classes to implement a different kind of linked-list implementation.

At each node of a doubly linked list, one link is a forward reference that points to the next node in the sequence, and the other link is a backward reference that points to the previous node in the sequence. As you can imagine, a doubly linked list is easy to traverse in either direction.

The .NET Framework provides a LinkedList class as part of the System.Collections.Generic namespace. This class implements a homogeneous doubly linked list of the specified data type. You can also write your own classes to implement a different kind of linked-list implementation

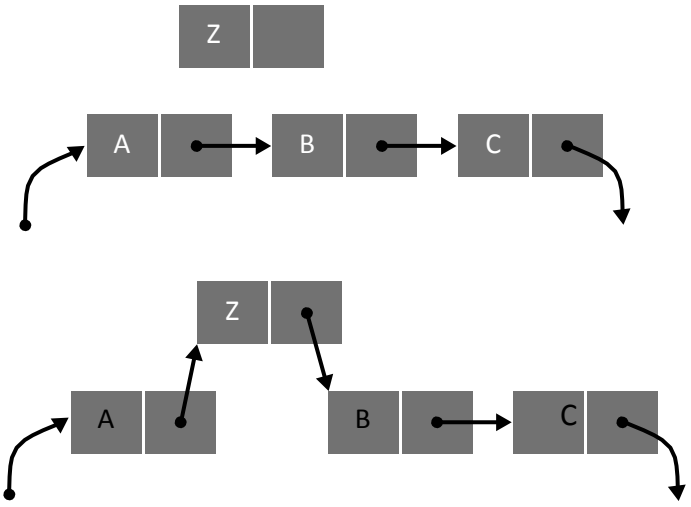
COMMON OPERATIONS

A linked list supports the following common operations:

- Add: Adding or inserting an item in a linked list is a matter of changing links, as shown in Figure 3-

6. Say you want to insert a new node (with value Z) between the nodes with values A and B. First, you need to allocate memory for the new node and assign value Z

Figure3-6



to the data section of the node. Next, you must copy the link section of node A to the link section of node Z so that node Z is pointing to node B. Finally, you must copy the address of the newly created node Z to the link section of node A so that node A starts pointing to node Z.

- Remove: Similar to the add operation, the remove or delete operation is also a mat- ter of changing links. For example, to delete the third node in Figure 3-4, you would change the link for the second node to a null value. The third node will now be an unreferenced piece of memory, and it will eventually be returned to the pool of avail- able memory.

- Find: The find operation finds a node with a given value in the linked list. To find a value, you generally start from the head node and check whether the value matches. If not, you follow the link to the next node and continue the find operation until you reach the end of the list, which happens when you encounter a null link.

PERFORMANCE AND USAGE

A linked list does not allow random access to its items. The only way to get to an item is to start from the head node and follow the links from there. As a result, linked lists are slow at retrieving data. However, for insert and delete operations, linked lists are extremely fast, because insertion or deletion of a node involves simply changing a link. Linked lists also have no maximum capacity after which their contents need to be reallocated.

In fact, a linked list provides an alternative way to implement the queue and the stack data structures. If your requirements call for frequent access to data but you seldom need to insert or delete data, an array

is the preferred implementation. If, however, your requirements call for frequent insert and delete operations, then a linked list may be a better implementation.

Understanding Sorting Algorithms

Sorting algorithms, such as BubbleSort and QuickSort, arrange items in a list in a particular order. Understanding sorting algorithms can help you understand, analyze, and compare different methods of problem solving.

Sorting algorithms are algorithms that arrange the items in a list in a certain order. For example, you can use a sorting algorithm to sort a list of students in ascending order of their last name. In the early days of data processing, sorting was an important problem that attracted a lot of research. These days, you can find basic sorting capabilities already built into most popular libraries and data structures. For example, in the .NET Framework, you can make use of the Array.Sort method to sort an array. However, it is still important to look at sorting as a way to understand problem solving and algorithm analysis.

In this section, you will take a look at two common sorting algorithms, BubbleSort and QuickSort

Understanding BubbleSort

The BubbleSort algorithm uses a series of comparison and swap operations to arrange the elements of a list in the correct order.

BubbleSort works by comparing two elements to check whether they are out of order; if they are, it swaps them. The algorithm continues to do this until the entire list is in the desired order. BubbleSort gets its name from the way the algorithm works: As the algorithm progresses, the smaller items are “bubbled” up.

Let’s visualize BubbleSort with the help of an example. Say you want to arrange all the items in the following list in ascending order: (20, 30, 10, 40). These items should be arranged from smallest to largest. The BubbleSort algorithm attempts to solve this problem in one or more passes, with each pass completely scanning the list of items. If the algorithm encounters out-of-order elements, it swaps them. The algorithm finishes when it scans the whole list without swapping any elements. If there were no swaps, then none of the elements were out of order and the list has been completely sorted.

Table 3-1

Step	Before	After	Comments
1	20, 30, 10, 40	20, 30, 10, 40	The algorithm compares the first two elements (20 and 30); because they are in the correct order, no swap is needed
2	20, 30, 10, 40	20, 10, 30, 40	The algorithm compares the next two elements (30 and 10); because they are out of order, the elements are swapped
3	20, 10, 30, 40	20, 10, 30, 40	The algorithm compares the next two elements (30 and 40); because they are in the correct order, no swap is needed

As shown in Table 3-1, at the end of first pass, BubbleSort has performed one swap, and there is the possibility that the items are not yet completely sorted. Therefore, BubbleSort gives the list another pass, as depicted in Table 3-2.

Table 3-2

Step	Before	After	Comments
1	20,10,30,40	10,20,30,40	The algorithm compares the first two elements (20 and 10); because they are out of order, the elements are swapped
2	10,20,30,40	10,20,30,40	The algorithm compares the next two elements (20 and 30); because they are in the correct order, no swap is needed.
3	10,20,30,40	10,20,30,40	The algorithm compares the next two elements (30 and 40); because they are in the correct order , no swap is needed

At the end of second pass, BubbleSort has performed one more swap, so it can't yet guarantee that the list is completely sorted. Thus, BubbleSort gives the list another pass, as

shown in Table 3-3.

Table 3.3

Step	Before	After	Comments
1	10,20,30,40	10,20,30,40	The algorithm compares the first two elements (10 and 20); because they are in the correct order, no swap is needed
2	10,20,30,40	10,20,30,40	The algorithm compares the next two elements (20 and 30); because they are in the correct order, no swap is needed
3	10,20,30,40	10,20,30,40	The algorithm compares the next two elements (30 and 40); because they are in the correct order, no swap is needed.

At the end of the third pass, BubbleSort didn't perform any swaps. . This guarantees that the list is now in sorted order and the algorithm can finish.

In C#, the BubbleSort algorithm can be expressed by the following method:

```
static int[] BubbleSort(int[] numbers)
{
bool swapped;
do
{
swapped = false;
for (int i = 0; i < numbers.Length - 1; i++)
{
if (numbers[i] > numbers[i + 1])
{
//swap
int temp = numbers[i + 1]; numbers[i + 1] = numbers[i]; numbers[i] =
temp;
swapped = true;
}
}
} while (swapped == true); return numbers;
}
```

Understanding QuickSort

The QuickSort algorithm uses the partitioning and comparison operations to arrange the elements of a list in the correct order.

The QuickSort algorithm uses the divide-and-conquer technique to continually partition a list until the size of the problem is small and hardly requires any sorting. The following steps explain this process in greater detail:

A list of size zero or one is always sorted by itself.

For a bigger list, pick any element in the list as a pivot element. Then, partition the list in such a way that all elements smaller than or equal to the pivot element go into the left list and all elements bigger than the pivot element go into the right list. Now, the combination of the left list, pivot element, and right list is always in sorted order if the left and the right list are in sorted order.

The problem is now partitioned into two smaller lists, the left list and the right list. Both these lists are solved using the technique described in the bullets above.Finally, all the small sorted lists are merged in order to create the final complete sorted list.

The following table explains the QuickSort algorithm with a brief example.
TABLE 3.4

Step	Data to be sorted							Comments
1	50,10,30,20,40							Start with an unsorted list and pick a pivot Element—in this case 30.
2	20 , 10			30	50,40			Partition the list, with items less than the pivot going to the left list and items greater than the pivot going to the right list. Then, to sort the left list, pick a pivot (here, 10). Similarly, to sort the right list, pick a pivot (here, 40) for that list
3	-	10	20	30	-	40	50	In the left list, 20 is greater than 10, and in the right list, 50 is greater than 40; therefore, both 20 and 50 go into the right list. This yields lists of only one number, which are all by definition sorted.
4	10,20,30,40,50							All the small sorted lists are merged to create the final complete sorted list.

So far, the main shortcoming of the QuickSort algorithm might appear to be the additional memory required by the creation of separate smaller lists. However, creating separate lists is not necessary. Using a slightly modified technique, the array can be partitioned in place, as shown in the following code listing:

```
static int Partition (int[] numbers, int left, int right, int pivotIndex)
{
int pivotValue = numbers[pivotIndex];
// move pivot element to the end int temp = numbers[right];
numbers[right] = numbers[pivotIndex]; numbers[pivotIndex] = temp;
// newPivot stores the index of the first
// number bigger than pivot int newPivot = left;
for (int i = left; i < right; i++)
{
if (numbers[i] <= pivotValue)
{
temp = numbers[newPivot]; numbers[newPivot] = numbers[i];
numbers[i] = temp;    newPivot++;
}
}

//move pivot element to its sorted position temp = numbers[right];
numbers[right] = numbers[newPivot]; numbers[newPivot] = temp;
return newPivot;
}
```

With this technique, first the pivot element is moved to the end of the array. Then, all the elements less than or equal to the pivot element are moved to the front of the array. Finally, the pivot element is placed just before the element greater than itself, effectively partitioning the array.

This partitioning algorithm can then be used by QuickSort to partition the list, reduce the problem to smaller problems, and recursively solve it:

```
static int[] QuickSort(int[] numbers, int left, int right)
{
if (right > left)
{
int pivotIndex = left + (right - left) / 2;
//partition the array pivotIndex = Partition(
numbers, left, right, pivotIndex);
// QuickSort
QuickSort(numbers,left,pivotIndex-1)
// sort the right partition QuickSort(
numbers, pivotIndex + 1, right);
}
return numbers
}
```

```
                }
```

Because of its partitioning approach, the QuickSort algorithm is much faster than the BubbleSort algorithm.